# Coincidental Correctness: An Interference or Interface to Successful Fault Localization?

Zheng Zheng, Yichao Gao, Peng Hao
School of Automation Science and Electrical Engineering,
Beihang University,
Beijing, China
zhengz@buaa.edu.cn

Zhenyu Zhang
Institute of Software
Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

***Abstract*—In software debugging, statistical fault localization techniques contrast dynamic spectra of program elements to estimate the location of faults in faulty programs. Coincidental correctness may have a negative impact on these techniques because faults can also be triggered in an observed non-failed run and thus disturbs the assessment of fault locations. However, eliminating the confounding relies on the accuracy of recognizing them. This paper makes use of the presence of coincidental correctness as an effective interface to the success of fault localization. We calculate the distribution overlapping of dynamic spectrum in failed runs and in non-failed runs to find out the *fault-leading* predicates, and further reduce the region by referencing the inter-class distances of the spectra to suppress the less suspicious candidates. Empirical results show that our technique can outperform representative existing predicate-based fault localization techniques.**

***Keywords—fault localization; coincidental correctness; class distribution***

## I. INTRODUCTION

Modern software and software systems are becoming more and more complicated, and both academia and industry require effective mechanisms to guarantee the quality of software. Most failed program runs are caused by faults existing in a program. Generally speaking, to fix a program fault, a developer needs to locate it first. Statistical fault localization (*SFL* in short) refers to the automatic process to locate suspicious program elements.

A SFL technique captures dynamic spectrum for each program element from the failed runs and the non-failed runs respectively, and contrasts them to estimate the *suspiciousness* of a program element being related to faults. Liblit et al. [10] proposed a scalable SFL technique CBI, which installs Boolean expressions (coined *predicates*) for specific program elements, and locates fault-relevant predicates to reduce the complexity of conventional SFL techniques that investigate every statement. To distinguish the two technique families, we refer to them as *predicate-based* SFL techniques and *statement-level* SFL techniques, respectively.

Although SFL techniques are reported successful, their effectiveness to locate faults is unavoidably influenced by the characteristics of input data. Coincidental correctness refers to the phenomenon that no failure is detected, even though the fault has been exercised [18]. The portion of non-failed runs that coincidentally manifest no abnormal behavior may have a negative impact on the accuracy of SFL techniques, because their execution profiles are closer to those of the failed runs (both with the fault triggered).

Previous studies have realized and validated the prevalence of coincidental correctness as well as its confounding to SFL techniques, and put efforts to address it. The direct idea is to recognize the coincidental correctness runs and remove them from inputs [6][12]. However, the feasibility and effectiveness are based on the accurate recognition of the coincidental correctness runs. The latest controlled experiment gave a pessimistic report that the false negative related to the recognition of coincidental correctness runs is above 50% for one out of three experiment subjects [13]. Can we allow the existence of coincidental correctness and locate a fault with the presence of it? The problem is both challenging and interesting.

In this paper, we analyze the behavior of dynamic spectra for different program predicates, with the presence of coincidental correctness, and propose a technique to find out the most fault-relevant predicates. First of all, we capture the dynamic spectra of program predicates in the failed runs and in the non-failed runs, respectively. After that, we calculate the overlapping of the spectrum distribution in the failed runs and that in the non-failed runs to find out the predicates, whose exercising lead to the triggering of a fault. Next, we reduce the region by calculating the inter-class distances for the spectra in the two communities (failed and non-failed) to suppress uninterested less suspicious predicates. We sort the predicates by referencing their calculated suspiciousness, and output a ranked list of suspicious predicates. Experiments show that our technique outperforms some representative existing predicate-based SFL techniques.

This paper makes the following contributions. (i) We propose a technique that properly estimates fault locations with the presence of coincidental correctness. It is expected to be more accurate since there is no longer a need to recognize the coincidentally correct runs. (ii) We use an empirical evaluation to show that our technique outperforms representative existing peer techniques on the common data sets.

The rest of this paper is organized as follows. Section II and III motivates and elaborates on our technique, respectively.

978-1-4799-2552-0/13/$31.00 ©2013 IEEE

| Program: `int max(int a, int b, int c) {` | Predicate | T1: (2,1,3) | T2: (2,1,2) | | T3: (1,1,3) | T4: (1,2,3) | | T5: (2,3,1) | T6: (1,2,0) | Observations |
|---|---|---|---|---|---|---|---|---|---|---|
| L1: `int x = a;` | P1: `x==a` | 0:1 | 0:1 | | 0:1 | 0:1 | | 0:1 | 0:1 | *Neutral* predicates: Similar spectra in all runs. |
| L2: `if (a > b)` | P2: `a>b` | 1:0 | 1:0 | ≠ | 0:1 | 0:1 | ≈ | 0:1 | 0:1 | *Fault-leading* predicates: |
| L3: `x = a;` | P3: `x==a` | 0:1 | 0:1 | ≠ | 0:0 | 0:0 | ≈ | 0:0 | 0:0 | Spectra in `{T3,T4}` resemble those in `{T5,T6}`, because both of them lead to trigger the fault; |
| L4: `else` | | | | | | | | | | spectra in `{T3,T4}` vary from those in `{T1,T2}`, |
| L5: `x = a; // x = b` | P4: `x==a` | 0:0 | 0:0 | ≠ | 0:1 | 0:1 | ≈ | 0:1 | 0:1 | because the latter lead to trigger no fault. |
| L6: | | | | | | | | | | |
| L7: `if (x > c)` | P5: `x>c` | 0:1 | 0:1 | ≈ | 0:1 | 0:1 | ≠ | 1:0 | 1:0 | *Fault-led* predicates: |
| L8: `return x;` | P6: `x==0` | 0:0 | 0:0 | ≈ | 0:0 | 0:0 | ≠ | 0:1 | 0:1 | Spectra in `{T3,T4}` vary from those in `{T5,T6}`, because the latter reveal failures; |
| L9: `else` | | | | | | | | | | spectra in `{T3,T4}` resemble those in `{T1,T2}`, |
| L10: `return c;` | P7: `c==0` | 0:1 | 0:1 | ≈ | 0:1 | 0:1 | ≠ | 0:0 | 0:0 | because both of them reveal no failure. |
| `}` | | successful | | | *coincidental* [a] | | | failed | | |

a. coincidental runs: they are also non-failed runs, but with coincidental correctness happening

Fig. 1. Motivating example (predicates may behave differently according to their relative position to fault)

Section IV and V give an empirical evaluation and a literature review, respectively. Section VI concludes the paper.

## II. MOTIVATION

### A. The Sample Program

Fig. 1 shows a piece of code to find the maximum value among three inputs. A fault is seeded in the statement `L5`, which may cause the program to generate an incorrect output.

We use simple integers as inputs to start the program and permute them to create six test cases, namely `T1`, `T2`, `T3`, `T4`, `T5`, and `T6`. We notice that although all of `T3`, `T4`, `T5`, and `T6` have exercised the faulty statement, only `T5` and `T6` generate unexpected outputs. We thus mark the program execution over `T5` and `T6` as failed runs and the other executions as non-failed runs, respectively. To ease the following discussion, we use the term *coincidental* runs, to name the program executions over `T3` and `T4`, where the fault is triggered but no faulty state is propagated to be finally observable. To differentiate from them, we use the term *successful* runs, to name the program runs over `T1` and `T2`.

Previous studies such as CBI construct predicates at three kinds of program statements (i.e., branch statements, return statements, and scalar-pairs). Following previous work [10], we install seven predicates in the program, and they are noted as `P1`, `P2`, `P3`, `P4`, `P5`, `P6`, and `P7` in Fig.1. We also record their dynamic spectrum in a program run in the form of $x : y$, where $x$ and $y$ stands for the number of times a predicate is evaluated to be true and the number of times that predicate is evaluated to be false, respectively. Let us take the first row to illustrate. In the program run of `T1`, predicate "`P1: x==a`" is evaluated false once and never evaluated true. We thus record the dynamic spectra of predicate `P1` as "0:1" in that run.

### B. Inspiring Our Work

Since the program contains no loop and is sequentially executed, we label three categories of predicates [1], namely *neutral* predicates, *fault-leading* predicates and *fault-led* predicates, according to their execution sequence in the failed runs. A rough judging is made according to the heuristics that (1) the exercising of a neutral predicate does not correlate with the exercising status of the fault, (2) the exercising of the fault-leading predicates may deterministically lead to trigger the fault or skip the fault, and (3) triggering the fault may deterministically lead to exercise the fault-led predicates or skip them. With such a criterion, the seven predicates are partitioned into three groups, in which `P1` is a neutral predicate, `P2`, `P3` and `P4` are fault-leading predicates, and `P5`, `P6`, and `P7` are fault-led predicates. Note that predicate `P4`, the most fault-relevant predicate, is also a fault-leading predicate with such classification. In Fig. 1, the observations are as below.

#### 1) On a Neutral Predicate

For a neutral predicate, its dynamic spectra in every runs resemble each other. This can be understood as that neutral predicates often have less relationship with the fault so that their behaviors make less difference in every runs no matter it is a failed run or a non-failed run. Let us take predicate `P1` in the program run `T4` to illustrate. Predicate "`P1: x==a`" lies on the first statement and always evaluates false, its spectra in all runs are equal.

#### 2) On a Fault-leading Predicate

For a fault-leading predicate, its dynamic spectra in the coincidental runs (the non-failed runs with coincidental correctness happening) are identical to those in the failed runs, but different from those in the successful runs (the non-failed runs without coincidental correctness happening). We use the symbols "≠" and "≈" in Fig. 1 to mark them for a better view. This can be understood as follows. The execution paths leading to a fault often concentrate into small clusters, as reported in [4]. Therefore, a fault-leading predicate may manifest similar dynamic spectra in the coincidental runs and the failed runs. Let us take predicate `P2` and the program run `T4` to illustrate. Predicate "`P2: a>b`" evaluates false, which skips the statement `L3` to triggers the fault on the statement `L5`. This is the *only* legitimate path that leads to the fault, and the dynamic

[1] Note that a predicate may have more than one label.

spectra for P2 in T4 are identical to those in T5 and T6, while is different from those in T1 and T2.

### 3) On a Fault-led Predicate

For a fault-led predicate, its dynamic spectra in the coincidental runs are identical to those in the successful runs, but different from those in the failed runs. This can be understood as follows. Even if the fault is exercised, the faulty state may be coincidentally not propagated and the led program executions still behave as normal. As a result, no difference can be observed in the dynamic spectra from the successful runs and those form the coincidental runs, for a fault-led predicate. Let us take predicate P5 and the program run T4 to illustrate. During the program run over T4, even with the faulty value of x, predicate "P5: x>c" gives a correct answer (i.e., it evaluates false). The fault is thus glossed over, which leads the rest program excerpt (from L7 to the end) to execute as normal. As a result, the dynamic spectra for P5 in T4 are identical to those in T1 and T2, while different from those in T5 and T6.

Such observations inspire us to utilize the spectrum distribution in different program runs to distinguish the fault-leading predicates from the others.

We find that the *overlapping* of spectrum distribution in failed runs with that in non-failed runs can be an effective means to differentiate a fault-leading predicate from the others. For the fault-leading predicate P2, its dynamic spectra in T5 and T6 are also observed in T3 and T4 (though not in T1 and T2), and we record an overlapping of 100% for it. The same phenomenon is observed with the predicates P3 and P4. For the fault-led predicate P5, its dynamic spectra in T5 and T6 are not observed in T1, T2, T3, and T4, and we record an overlapping of 0% for it. The same phenomenon is observed with the predicates P6 and P7. Thus, by comparing the extent of overlapping, we can rule out the fault-led predicates.

We have demonstrated that the overlapping of spectra in failed runs with that in non-failed runs can be helpful in indicating fault-leading predicates. However, to work out a successful fault localization technique, there are still challenges. First, a predicate can be evaluated more than once because of the presence of *loops*. What is the proper form for a dynamic spectrum? Second, we notice that the opposite overlapping (spectra in non-failed runs with that in failed runs) can be also a good indicator. How to scientifically assess the extent of the overlapping is a key problem. Third, the neutral predicates have high overlapping (e.g., 100% for P1 in the example) and they mix up with the fault-leading predicates. It is necessary to suppress their confounding in the result. How does our model work in such cases? We will elaborate on our model in the next section.

## III. OUR MODEL

### A. Problem Settings

Let $P$ be a faulty program with $m$ predicates, which are referred to as $p_1, p_2, \ldots, p_m$. Program runs $R = \{r_1, r_2, \ldots\}$ is partitioned into two sets, $N$ and $F$, where $N = \{n_1, n_2, \ldots, n_u\}$ is the set of $u$ non-failed runs and $F = \{f_1, f_2, \ldots, f_v\}$ is the set of $v$ failed runs. For example, in Fig. 1, $N = \{\mathrm{T1}, \mathrm{T2}, \mathrm{T3}, \mathrm{T4}\}$, and $F = \{\mathrm{T5}, \mathrm{T6}\}$.

In each program run, a predicate may be exercised or not, and the evaluation result can be passed or fail. We use the term $e_T(p_j, r_i)$ and the term $e_F(p_j, r_i)$ to record the number of times a predicate $p_j$ is evaluated true and the number of times it is evaluated false in the program run $r_i$, respectively.

Our aim is to estimate the suspicious predicates, which are most relevant to the faults causing the observed failed runs in $F$.

### B. Preliminaries

Before elaborating on our model, we first introduce some preliminaries. Following previous study [11], we use $x(p_j, r_i)$ to express the evaluation bias of a predicate $p_j$ in a program run $r_i$. Here, an evaluation bias is the probability of a predicate being evaluated true in a program run. It is calculated as $x(p_j, r_i) = \frac{e_T(p_j, r_i)}{e_T(p_j, r_i) + e_F(p_j, r_i)}$ [2]. We also use $x(p_j, f_i)$ and $x(p_j, n_i)$ to express the evaluation bias of a predicate $p_j$ in a failed run and a non-failed run, respectively.

Further, the vector $X_i^F = [x(p_1, f_i), x(p_2, f_i), \ldots, x(p_m, f_i)]$ is used to denote the vector of evaluation bias for each predicate in the $i$-th failed run $f_i$. Similarly, $X_i^T$ is used to denote the vector of evaluation bias in the $i$-th non-failed run $n_i$.

The *overlapping* of spectrum distribution in failed runs with that in non-failed runs actually considers the similarity between the two kinds of runs, according to the variables of evaluation bias which exist in both of them. In this paper, Bhattacharyya coefficient [2] is introduced to the measurement of the overlapping. It is a function to measure the amount of overlap between two statistical samples or populations. Let $y_j$ be a variable of evaluation bias for predicate $p_j$, and $\omega_F$, $\omega_N$ denote the failed runs and non-failed runs, respectively. Bhattacharyya coefficient can be formalized as

$$BC\left(\mathrm{P}(y_j|\omega_F), \mathrm{P}(y_j|\omega_N)\right) = \sum_{y_j \in D_j} \sqrt{\mathrm{P}(y_j|\omega_F) \times \mathrm{P}(y_j|\omega_N)}$$

where $D_j$ is the domain of $y_j$, $P(y_j|\omega_F)$ and $P(y_j|\omega_N)$ are the conditional probabilities of $y_j$ in the set of failed runs and non-failed runs respectively. The probability of $y_j$ exists in both of the two kinds of runs is denoted as $P(y_j|\omega_F) \times P(y_j|\omega_N)$. The measure is proved to be the upper bound of Bayes error, which directly related to the overlapping of two models [16]. In this paper, we use $\sigma_j^F/v$ and $\sigma_j^N/u$ to approximate $P(y_j|\omega_F)$ and $P(y_j|\omega_N)$ respectively, where $\sigma_j^F$ is the count of the appearance of $y_j$ in the set of failed runs, and $\sigma_j^N$ is the count of the appearance of $y_j$ in the set of non-failed runs. Take the predicate $p_1$ in Fig. 1 as an example, there exists only one variable $y_1 = 0$, so the conditional probabilities are $P(y_1|\omega_F)$ and $P(y_1|\omega_N)$ are 1, and $BC\left(\mathrm{P}(y_1|\omega_F), \mathrm{P}(y_1|\omega_N)\right) = 1$.

[2] Note that when the predicate is never evaluated, there is no clue to determine its evaluation bias value, and we follow previous work [11] to unbiasedly set it to 0.5.

## C. Our Technique

Our fault localization technique consists of four steps.

### 1) S1: Collecting Dynamic Spectra

In this step, the subject program is instrumented to log execution traces of predicates at runtime. We collect dynamic spectra for all predicates in every program runs, and predicates are inserted for three kinds of statements, that is, branch statements, scalar-pair statements, and return statements. We use the evaluation bias to capture the dynamic spectrum.

### 2) S2: Calculating the Overlapping of Spectrum Distributions

For neutral predicates and fault-leading predicates, since their spectra in coincidental runs and their spectra in failed runs resemble each other to a great extent, we adopt to estimate the distribution overlapping to differentiate them from the fault-led predicates. Bhattacharyya distance [2] is used to calculate the overlapping of spectrum distribution in failed runs with that in non-failed runs (note that we cannot figure out the coincidental runs from the non-failed runs).

Given a predicate $p_j$, the problem is to determine the overlapping of its spectrum distribution in failed runs $F$ and that in non-failed runs $N$. The overlapping $O_j$ of the spectrum distribution in failed runs with that in non-failed runs can be expressed in terms of the Bhattacharyya distance:

$$O_j = -\ln\left[BC\left(P(y_j|\omega_F), P(y_j|\omega_N)\right)\right],$$

where $BC(\cdot)$ is the Bhattacharyya coefficient. If $BC\left(P(y_j|\omega_N), P(y_j|\omega_F)\right) = 0$, we set $O_j$ to be $+\infty$.

After this step, we may reorder all the predicates by referencing their overlapping value in the descending order. The top ranked predicates is supposed to contain more fault-leading predicates. However, we also predict that after such a step, the neutral predicates may still mix up with the fault-leading predicates in the results.

### 3) S3: Calculating the Inter- and Intra-class Distances

We notice that the neutral predicates still mix up with the fault-leading predicates. Let us focus on the inter-class distance [7] to figure out a solution. In the motivating example, we have demonstrated that for fault-leading and fault-led predicates, their spectra in successful runs and in failed runs are different from each other to a great extent. We thus adopt to estimate the inter-class distance to differentiate them from the neutral predicates.

The inter-class distance $B_j$ for a predicate $p_j$ is calculated as:

$$B_j = \left|m_j^F - m_j^N\right|$$

where $m_j^F$ and $m_j^N$ are the mean value of the evaluation bias for $p_j$ in the failed and the non-failed program runs, respectively. Here, $m_j^F$ and $m_j^N$ are calculated as follow.

$$m_j^F = \frac{1}{v}\sum_{i=1}^{v}\left[x(p_j, f_i)\right], \quad m_j^N = \frac{1}{u}\sum_{i=1}^{u}\left[x(p_j, n_i)\right]$$

The inter-class distance $B_j$ captures the distance between the evaluation bias of predicate $p_j$ in the set of failed runs and the evaluation bias of it in the set of non-failed runs. As in the motivation, we have explained that the inter-class distance for a neutral predicate is less than that of a fault-leading predicate. Thus we can use $B_j$ to differentiate a neutral predicate from a fault-leading predicate.

However, we also realize that spectrum distributions for two predicates may have unequal widths. Directly comparing their inter-class distance may not be scientific. For example, the predicate installed for the branch statement of a long loop may have very small evaluation bias value[3]. The inter-class distance calculated for it can be much smaller than the average. To fairly compare the inter-class distance of two predicates, we further reference their intra-class distance to normalize them before comparison.

The intra-class distance $D_j$ for a predicate $P_j$ is calculated as,

$$D_j = \frac{\sqrt{\frac{\sum_{i=1}^{v}\left[\left(x(p_j, f_i) - m_j^F\right)^2\right]}{v}} + \sqrt{\frac{\sum_{i=1}^{u}\left[\left(x(p_j, n_i) - m_j^N\right)^2\right]}{u}}}{2}$$

It can be similarly explained as $B_j$. Note that it is the mean of the intra-class distance of $P_j$ for the failed runs and that for the non-failed runs.

We normalize the inter-class distance $B_j$ using the intra-class distance $D_j$ for each predicate, so that their distance can be fairly compared to each other. The normalized inter-class distance $A_j$ for $p_j$ is as follows.

$$A_j = \frac{B_j}{D_j}$$

When $D_j$ is zero and $B_j$ is not zero, we set $A_j$ to be $+\infty$. When $D_j$ is zero and $B_j$ is also zero, we set $A_j$ to be zero.

This step decreases the ranks of the neutral predicates without affecting the relative order of the fault-leading predicates and the fault-led predicates.

### 4) S4: Generating a Ranked List of Suspicious Predicates

In previous step, we use the normalized inter-class distance $A_j$ to differentiate a neutral predicates from a fault-leading predicates. By integrating the two steps, we have the suspiciousness formula $S_j$ as follows,

$$S_j = 2^{(O_j - A_j)}.$$

Since the use of $O_j$ can rule out the fault-led predicates, and the use of $A_j$ can suppress neutral predicates, we thus identify fault-leading predicates. At the same time, since the normalized inter-class distance for a fault-leading predicate is supposed to be comparable to that of a fault-led predicate, the relative order of fault-leading predicates and fault-led predicates is still reserved by the adjustment of "$-A_j$". The base number 2 is to assure that $S_j \geq 0$.

Finally, we reorder the predicates in the descending order of their suspiciousness scores $S_j$, and generate a ranked list of predicates.

[3] E.g., predicate "`i<9`" in "`for(i=0; i<9; i++)`" always has an evaluation bias of 0.1.

TABLE I. STATISTICS OF SUBJECT PROGRAMS IN USED

| Programs | # of selected versions | # of LOC | # of predicates | # of runs |
|---|---|---|---|---|
| print_tokens | 4 | 472 | 51 | 4130 |
| print_tokens2 | 10 | 399 | 116 | 4115 |
| schedule | 9 | 292 | 24 | 2650 |
| schedule2 | 9 | 301 | 55 | 2710 |
| replace | 30 | 512 | 63 | 5542 |
| tot_info | 19 | 440 | 47 | 1052 |
| tcas | 30 | 141 | 10 | 1608 |
| space | 28 | 6218 | 914 | 13585 |
| flex | 20 | 15297 | 895 | 567 |
| grep | 12 | 15633 | 1284 | 809 |

## IV. EMPIRICAL EVALUATION

In this section we conduct experiments to test the effectiveness of our method. We describe the experiment setup, including the subject programs, peer techniques, and the effectiveness metrics for fault localization. The results and analysis of our experiments are presented subsequently. Finally, we discuss the threats to validity of our experiment.

### A. Experiment Design

#### 1) Subject programs

To evaluate our technique, we use the Siemens suite [5], a realistic program space, and two UNIX programs flex and grep as experiment subjects (see TABLE I). We excluded the versions come with no failed run or having a failure rate greater than 20% [21]. 171 faulty versions are used in our experiments.

#### 2) Peer techniques

To adequately evaluate our method, we compare it with the predicate-based techniques CBI [10], SOBER [11], Wilcoxon [20], and Mann-Whitney [20]. We choose the former two because they are representative. We choose the latter two because they show promising results in a last report [20].

We do not select any popular statement-level techniques for comparison due to three considerations. (i) It is not fair to directly compare the effectiveness of a predicate-based technique with that of a statement-level technique. (ii) We want to focus on predicate-based techniques to consistently evaluate our method. (iii) The Wilcoxon and Mann-Whitney techniques have been empirically shown comparable to the state-of-the-art statement-level techniques [20].

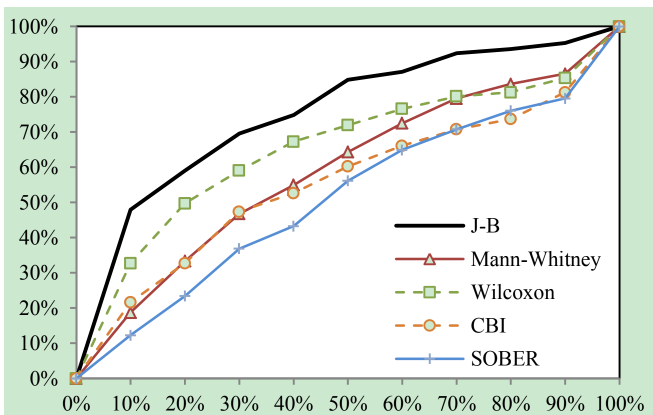


Fig. 2. Overall effectiveness comparison

#### 3) Effectiveness metrics

P-score [20] uses the appearance position of the most fault-relevant predicate in the generated ranked list of predicates as the effectiveness of that fault-localization technique to locate a fault.

The effectiveness for our technique is consistently coined as "J-B" in the rest part of this section. The other techniques are referred to by their names.

### B. Results

Fig. 2 depicts the overall effectiveness of each technique. The x-axis of Fig. 2 shows the predicate examination efforts (the percentage of predicates examined). The y-axis of Fig. 2 shows the percentage of faults located (P-score) within the given predicate examination efforts. The curve of J-B, which stands for our technique, is shown in bold, and curves for other techniques are shown with different colors and markers.

Fig. 2 shows that, for the 171 faulty versions in all the programs, our technique always locates more faults than the other techniques, when using any predicate examination effort of 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% as a check point. For example, when a developer examines at most 10% of the predicates, CBI and SOBER can catch faults in 21.64% and 12.28% of the faulty versions, respectively, while Wilcoxon and Mann-Whitney capture faults in 32.75% and 18.71% faulty versions, respectively. With the same predicate examination efforts, our method can locate faults in 47.95% faulty versions. The advantages of our technique are observable. In summary, Fig. 2 shows that our technique has an overall advantage to the other techniques in locating faults in the subject programs.

### C. Threats to Validity

In our technique, Bhattacharyya distance and inter-class distance [7] are used to evaluate the spectrum distribution overlapping and the inter-class distance, respectively. Since both the overlapping and the inter-class distance are evaluated using symmetric metrics, other popular distance measurements can be also adopted. They may result in different experiment observations and conclusions. On the other hand, effectiveness metric may cause threats to the construct validity of the results.

External validity of the experiment can be threatened by the use of other subject programs. The experimental observation may be consolidated by using more subjects in evaluation.

Threats to the validity of the experiment also relate to the impact factors of the experiment conclusions. In our technique, we employ two different measurements in two steps. We predict that by using each of them independently we cannot achieve the desired results. Though the experiment also gives positive answers, whether one of them has a dominant effect is unknown. Orthogonal experiments to validate the net effect of each step may give more insights on our technique.

## V. Related Work

Tarantula [9] is one of the most famous fault localization techniques. It uses the proportions of failed or passed executions that exercising a statement to calculate the suspiciousness of that statement. Naish et al. [14] gave a summary for such techniques.

Compared to such statement-level techniques, CBI [10] uses predicates as fault indicators to locate faults, which gains both low complexity and high extensibility. Zhang et al. [22] empirically validated that the short-circuiting rule to evaluate a Boolean expression has significantly effects on the predicate-based techniques, and proposed DES [22] accordingly. Zhang et al. [20] proposed a non-parametric predicate-based statistical fault-localization framework. Arumuga Nainar et al. [1] further used compound Boolean predicates to locate faults. HOLMES [3] uses execution path as a fault predicator. Our technique can be also applied by using paths as predicates. In the future work, we will make further investigation. Other related work includes CP [20] and [15], which uses execution spectra of control flow edges to locate faults.

Coincidental correctness is a well-known impact factor of statistical fault localization. It causes program runs, which trigger the fault, to be marked as non-failed runs. Test suite reduction is a solution [8][15] to address coincidental correctness or improve test suite quality [19], but its feasibility relies on the accuracy of recognizing coincidental cases [13]. This paper proposes a methodology to address coincidental correctness, which does not rely on the accuracy of recognizing them. In this paper, Bhattacharyya coefficient is used to measure the similarity of the predicate spectra between failed runs and non-failed runs, to rule out the fault-led predicates. The inter- and intra-class distances are often used in pattern recognition to measure the class difference [16][17]. In this work, we use them to pick out the neutral predicates which mix up with the fault-leading predicates. The fundamental difference of this work with the mentioned previous studies is that it utilizes the presence of coincidental correctness as an effective interface to successful fault localization, rather than to eliminate it from inputs and propose a yet another suspiciousness metrics.

## VI. Conclusion

Oracle used in real-life can be seldom perfect, and the use of imperfect oracles in fault localization causes the prevalent coincidental correctness in practice. A popular approach is to get rid of them, since their presence make the input data of statistical fault localization techniques unreliable. However, its feasibility relies on the accuracy of recognizing them.

In this paper, we propose to utilize the presence of coincidental correctness to locate faults. We analyze the confounding of coincidental correctness to fault localization, propose to measure the spectrum distribution overlapping to rule out fault-led predicates, and further suppress neutral predicates by assessing the inter-class distribution of spectrum in the failed runs and the non-failed runs. A preliminary evaluation shows that our technique can more effectively locate the fault-leading predicates that are tightly related to faults, compared to the other representative techniques.

Future work includes involving execution path as additional information to accurately identify fault-leading predicates, and validating the proposed idea using statement-level techniques.

## References


[1] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, Statistical debugging using compound Boolean predicates, *Proc. ISSTA* ,pp. 5-15, 2007.

[2] A. Bhattacharyya. On a measure of divergence between two statistical populations defined by probability distributions. *Bulletin of the Calcutta Mathematical Society*, 1943.

[3] T.M. Chilimbi, B. Liblit, K. Mehra, A.V. Nori, and K. Vaswani, HOLMES: effective statistical debugging via efficient path profiling, *Proc. ICSE,* pp. 34-44, 2009.

[4] W. Dickinson, D. Leon, and A. Podgurski. Pursuing failure: the distribution of program failures in a profile space. *Proc. ESEC/FSE,* pp. 246-255, 2001.

[5] H. Do, S. G. Elbaum, and G. Rothermel, Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact, *Experimentation in Software Engineering*, vol. 10(4), pp. 405-435, 2005.

[6] R. Gore, and P. F. Reynolds. Reducing confounding bias in predicate-level statistical debugging metrics, *Proc. ICSE,* pp. 463-473, 2012.

[7] Y. Guan, H. Wang. Set-valued information systems. *Information Sciences*, vol. 176(17), pp. 2507-2525, 2006.

[8] D. Hao, Y. Pan, L. Zhang, W. Zhao, H. Mei and J. Sun, A similarity-aware approach to testing based fault localization, *Proc. ASE,* pp. 291-294, 2005.

[9] J.A. Jones and M.J. Harrold, Empirical evaluation of the Tarantula automatic fault-localization technique, *Proc. ASE,* pp. 273-282, 2005.

[10] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M.I. Jordan, Scalable statistical bug isolation, *Proc. PLDI,* pp. 15-26, 2005.

[11] C. Liu, L. Fei, X. Yan, S. P. Midkiff, and J. Han, Statistical debugging: a hypothesis testing-based approach, *IEEE TSE*, vol. 32(10), pp. 831-848, 2006.

[12] W. Masri and R. A. Assi. Cleansing test suites from coincidental correctness to enhance fault-localization, *Proc. ICST,* pp. 165-174, 2010.

[13] Y. Miao, Z. Chen, S. Li, Z. Zhao, and Y. Zhou, Identifying coincidental correctness for fault localization by clustering test cases, *Proc. SEKE,* pp. 262-272, 2012.

[14] L. Naish, H.J. Lee, and K. Ramamohanarao, A model for spectra-based software diagnosis, *ACM TOSEM*, vol. 20(3):11, 2011.

[15] R. Santelices, J.A. Jones, Y. Yu, and M.J. Harrold, Lightweight fault-localization using multiple coverage types, *Proc. ICSE,* pp. 56-66, 2009.

[16] S. Theodoridis, K. Koutroumbas. *Pattern Recognition*. Academic Press, New York, 4th. 2009.

[17] L. Wang, Feature selection with kernel class separability, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30(9), pp. 1534-1546, 2008.

[18] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. *Proc. ICSE,* pp. 45-55, 2009.

[19] Y. Yu, J. A. Jones, and M.J. Harrold, An empirical study of the effects of test-suite reduction on fault localization, *Proc. ICSE,* pp. 201-210, 2008.

[20] Z. Zhang, W.K. Chan, T.H. Tse, B. Jiang, and X. Wang, Capturing propagation of infected program states, *Proc. ESEC/FSE,* pp.43-52, 2009 .

[21] Z. Zhang, W. K. Chan, T. H. Tse, Y. T. Yu, and P. Hu, Non-parametric statistical fault localization, *Journal of Systems and Software*, vol. 84(6), pp. 885-905, 2011.

[22] Z. Zhang, B. Jiang, W. K. Chan, T. H. Tse, and X. Wang, Fault localization through evaluation sequences, *Journal of Systems and Software*, vol. 84(6), 2010.